One Sided Communication

- One sided communication allows shmem style gets and puts
- Only one process need actively participate in one sided operations
- With sufficient hardware support, remote memory operations can offer greater performance and functionality over the message passing model
- MPI remote memory operations do not make use of a shared address space
- One sided comms are sensitive to OS/machine optimizations though

One Sided Communication

- By requiring only one process to participate, significant performance improvements are possible
  - No implicit ordering of data delivery
  - No implicit synchronization
- Some programs are more easily written with the remote memory access (RMA) model
  - Global counter

Standard message passing

Packet transmission is directly mitigated by the CPU's on both machines, multiple buffer copies may be necessary
Traditional message passing
- Both sender and receiver must cooperate
  - Send needs to address buffer to be sent
  - Sender specifies destination and tag
  - Recv needs to specify it's own buffer
  - Recv must specify origin and tag
- In blocking mode this is a very expensive operation
  - Both sender and receiver must cooperate and stop any computation they may be doing

Sequence of operations to `get` data
- Suppose process A wants to retrieve a section of an array from process B (process B is unaware of what is required)
  - Process A executes MPI_Send to B with details of what it requires
  - Process executes MPI_Recv from A and determines data required by A
  - Process B executes MPI_Send to A with required data
  - Process A executes MPI_Recv from B...
- 4 MPI-1 commands
- Additionally process B has to be aware of incoming message
  - Requires frequent polling for messages – potentially highly wasteful

Even worse example
- Suppose you need to read a remote list to figure out what data you need – sequence of ops is then:

  Process A
  MPI_Send (get list)
  MPI_Send (list returned)
  MPI_Send (get data)
  MPI_Recv (data returned)

  Process B
  MPI_Recv (list request)
  MPI_Send (list info)
  MPI_Recv (data request)
  MPI_Send (data info)

Coarse versus fine graining
- Expense of message passing implicitly suggests MPI-1 programs should be coarse grained
- Unit of messaging in NUMA systems is the cache line
  - What about API for (fast network) distributed memory systems that is optimized for smaller messages?
    - e.g. ARMCI http://www.emsl.pnl.gov/docs/parsoft/armci
  - Would enable distributed memory systems to have moderately high performance fine grained parallelism
  - A number of applications are suited to this style of parallelism (especially irregular data structures)
  - T3E and T3D both capable of performing fine grained calculations – well balanced machines
  - API’s supporting fine grained parallelism have one-sided communication for efficiency – no handshaking to take processes away from computation
Puts and Gets in MPI-2

- In one sided communication the number of operations is reduced by (at least) a factor of 2
  - If communication patterns are dynamic and unknown then four MPI operations may be replaced by one MPI_Get/Put
- Circumvents the need to forward information directly to the remote CPU specifying what data is required
- MPI_Sends+MPI_Recv's are replaced by three possibilities
  - MPI_Get: Retrieve section of a remote array
  - MPI_Put: Place a section of a local array into remote memory
  - MPI_Accumulate: Remote update over operator and local data
- However, programmer must be aware of the possibility of remote processes changing local arrays!

Benefits of one-sided communication

- No matching operation required for remote process
- All parameters of the operations are specified by the origin process
- Allows very flexible communications patterns
  - Communication and synchronization are separated
    - Synchronization is now implied by the access epoch
- Removes need for polling for incoming messages
- Significantly improves performance of applications with irregular and unpredictable data movement

Windows: The fundamental construction for one-sided comms

- One sided comms may only write into memory regions (“windows”) set aside for communication
- Access to the windows must be within a specific access epoch
- All processes may agree on access epoch, or just a pair of processes may cooperate
Creating a window

- `MPI_Win_create(base, size, disp_unit, info, comm, win, ierr)`
  - Base address of window
  - Size of window in BYTES
  - Local unit size for displacements (BYTES, e.g. 4)
  - Info – argument about type of operations that may occur on window
  - Win – window object returned by call
- Should also free window using `MPI_Win_free(win, ierr)`
- Window performance is always better when base aligns on a word boundary

Options to info

- Vendors are allowed to include options to improve window performance under certain circumstances
- `MPI_INFO_NULL` is always valid
- If `win_lock` is not going to be used then this information can be passed as an info argument:

  ```
  MPI_Info info;
  MPI_Info_create(&info);
  MPI_Info_set(info, "no_locks", "true");
  MPI_Win_create(…, info,…);
  MPI_Info_free(&info);
  ```

Rules for memory areas assigned to windows

- Memory regions for windows involved in active target synchronization may be statically declared
- Memory regions for windows involved in passive target access epochs may have to be dynamically allocated
  - depends on implementation
  - For Fortran requires definition of Cray-like pointers to arrays
  - `MPI_Alloc_mem(size, MPI_INFO_NULL, baseptr)`
  - Must be associated with freeing call
    - `MPI_Free_mem(baseptr)`

Access epochs

- Although communication is mediated by GETs and PUTs they do not guarantee message completion
- All communication must occur within an access epoch
- Communication is only guaranteed to have completed when the epoch is finished
  - This is to optimize messaging – do not have to worry about completion until access epoch is ended
- Two ways of coordinating access
  - Active target: remote process governs completion
  - Passive target: Origin process governs completion
Access epochs: Active target

- Active target communication is usually expressed in a collective operation.
- All processes agree on the beginning of the window.
- Communication occurs.
- Communication is then guaranteed to have completed when second `WIN_FENCE` is called.

Access epochs: Passive target

- For passive target communication, the origin process controls all aspects of communication.
- Target process is oblivious to the communication epoch.
- `MPI_Win_(un)lock` facilitates the communication.

Cray SHMEM – origin of many one-sided communication concepts

- On the T3E, a number of variable types were guaranteed to occupy the same point in memory on different nodes:
  - Global variables/variables in common blocks
  - Local static variables
  - Fortran variables specified via `!DIR$ SYMMETRIC` directive
  - C variables specified by `#pragma symmetric` directive
  - Variables that are stack allocated, or dynamically on the heap are not guaranteed to occupy the same address on different processors.
- These variables could be rapidly retrieved/updated via `shmem_get/put`.
- `MPI_Win_fence`!
- `MPI_Win_fence`
- `MPI_Win_fence`

MPI_Get/Put/Accumulate

- Non-blocking operations.
- `MPI_Get(origin address,count,datatype,target,target displ,target target count,datatype,win,ierr)`
  - Must specify information about both origin and remote datatypes – more arguments
  - No need to specify communicator – contained in window
  - `Target displ` is displacement from beginning of target window.
  - `Note remote datatype cannot resolve to overlapping entries`.
- `MPI_Put` has same interface.
- `MPI_Accumulate` requires the reduction operator also be specified (argument before the window).
  - Same operators as `MPI_REDUCE`, but user defined functions cannot be used.
  - Note `MPI_Accumulate` is really `MPI_Put_accumulate`, there is no get functionality (must do by hand).
MPI_Accumulate

- Extremely powerful operation “put+op”
- Question marks for implementations though
  - Who actually implements the “op” side of things?
  - If on remote node then there must be an extra thread to do this operation
  - If on local node, then accumulate becomes get followed by operation followed by put
- Many computations involve summing values into fields
  - MPI_Accumulate provides the perfect command for this
- For scientific computation it is frequently more useful than MPI_Put

Don’t forget datatypes

- In one-sided comms datatypes play an extremely important role
  - Specify explicitly the unpacking on the remote node
  - Origin node must know precisely what the required remote data type is

Use PUTs rather than GETs

- Although both PUTs and GETs are non-blocking it is desirable to use PUTs whenever possible
  - GETs imply an inherent wait for data arrival and only complete when the message side has fully decoded the incoming message

MPI_Win_fence

- MPI_Win_fence(info,win,ierr)
  - Info allows user to specify constant that may improve performance (default of 0)
    - MPI_MODE_NOSTORE: No local stores
    - MPI_MODE_NOPUT: No puts will occur within the window (don’t have to watch for remote updates)
    - MPI_MODE_NOPRECEDE: No earlier epochs of communication (optimize assumptions about window variables)
    - MPI_MODE_NOSUCCEED: No epochs of communication will follow this fence
    - NO_PRECEDE and NOSUCCEED must be called collectively
- Multiple messages sent to the same target between fences may be concatenated to improve performance
**MPI_Win_(un)lock**

- **MPI_Win_lock(lock_type,target,info,win,ierr)**
  - Lock_types:
    - MPI_LOCK_SHARED – use only for concurrent reads
    - MPI_LOCK_EXCLUSIVE – use when updates are necessary
  - Although called a lock – it actually isn’t (very poor naming convention)
    - “MPI_begin/end_passive_target_epoch”
    - Only on the local process does MPI_Win_lock act as a lock
    - Otherwise non-blocking
  - Provides a mechanism to ensure that the communication epoch is completed
  - Says nothing about order in which other competing message updates will occur on the target (consistency model is not specified)

**Problems with passive target access**

- Window creation must be collective over the comm
  - Expensive and time consuming
- MPI_Alloc_mem may be required
- Race conditions on a single window location under concurrent get/put must be handled by user
- Local and remote operations on a remote window cannot occur concurrently even if different parts of the window are being accessed at the same time
- Local processes must execute MPI_Win_lock as well
- Multiple windows may have overlap, but must ensure concurrent operations to do different windows do not lead to race conditions on the overlap
- Cannot access (via MPI_get for example) and update (via a put back) the same location in the same access epoch (either between fences or lock/unlock)

**Drawbacks of one sided comms in general (slightly dated)**

- No evidence for advantage except on
  - SMP machines
  - Cray distributed memory systems (and Quadrics and now Infiniband)
    - Although advantage on these machines is significant – on T3E MPI latency is 16 µs, SHMEM latency is 2 µs
- Slow acceptance
  - Myrinet one sided comms “coming soon”
  - MPICH2 still not in full release
  - LAM supports only active target
- Unclear how many applications actually benefit from this model
  - Not entirely clear whether nonblocking normal send/recvs can achieve similar speed for some applications

**Hardware – Reasons to be optimistic**

- Newer network technologies (e.g. Infiniband, Quadrics) have a built in RDMA engine
  - RMA framework can built on top of the NIC library (“verbs”)
- 10 gigabit ethernet will almost certainly come with an RDMA engine
- Myrinet and SCI will both have one sided comms implemented very soon (after years of procrastination)
- Still in its infancy – number of software issues to work out
  - Support for non-contiguous datatypes is proving difficult – need efficient way to deal with the gather/scatter step
  - Many RDMA engines are designed for movement of contiguous regions – a comparatively rare operation in many situations
  - See http://nowlab.cis.ohio-state.edu/projects/mpi-iba/
Case Study: Matrix transpose

• See Sun documentation

• Need to transpose elements across processor space
  – Could do one element at a time (bad idea!)
  – Aggregate as much local data as possible and send large message (requires a lot of local data movement)
  – Send medium-sized contiguous packets of elements (there is some contiguity in the data layout)

Program 1

```fortran
include "mpif.h"
real(8), allocatable, dimension(:) :: a, b, c, d
real(8) t0, t1, t2, t3
!
! initialize parameters
!
! allocate matrices
!
! timing
! first local transpose
!
! global all-to-all
!
! second local transpose
!
! check
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
Summary

- One sided comms can reduce synchronization and thereby increase performance
- They indirectly reduce local data movement
- The reduction in messaging overhead can simplify programming